dijitso Documentation

Release 2018.1.0

FEniCS Project

Jul 26, 2018

Contents

A Python module for distributed just-in-time shared library building.

DIJITSO is part of the FEniCS Project.

For more information, visit http://www.fenicsproject.org

CHAPTER 1

Documentation

1.1 Installation

DIJITSO is normally installed as part of an installation of FEniCS. If you are using DIJITSO as part of the FEniCS software suite, it is recommended that you follow the installation instructions for FEniCS.

To install DIJITSO itself, read on below for a list of requirements and installation instructions.

1.1.1 Requirements and dependencies

DIJITSO requires Python version 2.7 or later and depends on the following Python packages:

- NumPy

These packages will be automatically installed as part of the installation of DIJITSO, if not already present on your system.

Additionally, to run tests the following packages are needed

- pytest
- mpi4py (for running tests with mpi)

1.1.2 Installation instructions

To install DIJITSO, download the source code from the DIJITSO Bitbucket repository, and run the following command:

```
pip install .
```

To install to a specific location, add the `--prefix` flag to the installation command:

```
pip install --prefix=<some directory> .
```

1.1.3 Environment

Instant's behaviour depened on following environment variables:

- `DIJITSO_CACHE_DIR`

  This option overrides the placement of the cache directory. By default the cache directory is placed in `.cache/dijitso` either below the home directory or below the prefix of the currently active virtualenv or conda environment if any.

- `DIJITSO_SYSTEM_CALL_METHOD`

  Choose method for calling external programs (c++ compiler). Available values:

  - `SUBPROCESS`

    Uses pipes. OFED-fork safe on Python 3. Default.

  - `OS_SYSTEM`

    Uses temporary files. Probably OFED-fork safe.

Warning: OFED-fork safe system call method might be required to avoid crashes on OFED-based (InfiniBand) clusters!

1.2 User manual

The commandline tool 'dijitso' is currently only documented on the commandline, run 'dijitso –help' for details and available subcommands.

The python module dijitso is only used internally by ffc, and no manual has been written so far. See API reference.

1.3 dijitso package

1.3.1 Submodules

1.3.2 dijitso.build module

Utilities for building libraries with dijitso.

`dijitso.build.`**`build_shared_library`**(*signature*, *header*, *source*, *dependencies*, *params*)
Build shared library from a source file and store library in cache.

`dijitso.build.`**`make_compile_command`**(*src_filename*, *lib_filename*, *dependencies*, *build_params*, *cache_params*)
Piece together the compile command from build params.

Returns the command as a list with the command and its arguments.

`dijitso.build.`**`make_unique`**(*dirs*)
Take a sequence of hashable items and return a tuple including each only once.

Preserves original ordering.

dijitso.build.**temp_dir**(*cache_params*)
Return a uniquely named temp directory.

Optionally residing under temp_dir_root from cache_params.

1.3.3 dijitso.cache module

Utilities for disk cache features of dijitso.

dijitso.cache.**analyse_load_error**(*e*, *lib_filename*, *cache_params*)

dijitso.cache.**check_cache_integrity**(*cache_params*)
Check dijitso cache integrity.

dijitso.cache.**clean_cache**(*cache_params*, *dryrun=True*, *categories=('inc', 'src', 'lib', 'log')*)
Delete files from cache.

dijitso.cache.**compress_source_code**(*src_filename*, *cache_params*)
Keep, delete or compress source code based on value of cache parameter 'src_storage'.

Can be "keep", "delete", or "compress".

dijitso.cache.**create_fail_dir_path**(*signature*, *cache_params*)
Create path name to place files after a module build failure.

dijitso.cache.**create_inc_basename**(*signature*, *cache_params*)
Create header filename based on signature and params.

dijitso.cache.**create_inc_filename**(*signature*, *cache_params*)
Create header filename based on signature and params.

dijitso.cache.**create_lib_basename**(*signature*, *cache_params*)
Create library filename based on signature and params.

dijitso.cache.**create_lib_filename**(*signature*, *cache_params*)
Create library filename based on signature and params.

dijitso.cache.**create_libname**(*signature*, *cache_params*)
Create library name based on signature and params, without path, prefix 'lib', or extension '.so'.

dijitso.cache.**create_log_filename**(*signature*, *cache_params*)
Create log filename based on signature and params.

dijitso.cache.**create_src_basename**(*signature*, *cache_params*)
Create source code filename based on signature and params.

dijitso.cache.**create_src_filename**(*signature*, *cache_params*)
Create source code filename based on signature and params.

dijitso.cache.**ensure_dirs**(*cache_params*)

dijitso.cache.**extract_files**(*signature*, *cache_params*, *prefix=''*, *path='.'*, *categories=('inc', 'src', 'lib', 'log')*)
Make a copy of files stored under this signature.

Target filenames are '<path>/<prefix>-<signature>.*'

dijitso.cache.**extract_function**(*lines*)
Extract function code starting at first line of lines.

dijitso.cache.**extract_lib_signatures**(*cache_params*)
Extract signatures from library files in cache.

dijitso.cache.**get_dijitso_dependencies**(*libname*, *cache_params*)
Run ldd and filter output to only include dijitso cache entries.

dijitso.cache.**glob_cache**(*cache_params*, *categories=('inc', 'src', 'lib', 'log')*)
Return dict with contents of cache subdirectories.

dijitso.cache.**grep_cache**(*regex*, *cache_params*, *linenumbers=False*, *countonly=False*, *signature=None*, *categories=('inc', 'src', 'log')*)
Search through files in cache for a pattern.

dijitso.cache.**load_library**(*signature*, *cache_params*)
Load existing dynamic library from disk.

Returns library module if found, otherwise None.

If found, the module is placed in memory cache for later lookup_lib calls.

dijitso.cache.**lookup_lib**(*lib_signature*, *cache_params*)
Lookup library in memory cache then in disk cache.

Returns library module if found, otherwise None.

dijitso.cache.**make_inc_dir**(*cache_params*)

dijitso.cache.**make_lib_dir**(*cache_params*)

dijitso.cache.**make_log_dir**(*cache_params*)

dijitso.cache.**make_src_dir**(*cache_params*)

dijitso.cache.**read_inc**(*signature*, *cache_params*)
Lookup header file in disk cache and return file contents or None.

dijitso.cache.**read_library_binary**(*lib_filename*)
Read compiled shared library as binary blob into a numpy byte array.

dijitso.cache.**read_log**(*signature*, *cache_params*)
Lookup log file in disk cache and return file contents or None.

dijitso.cache.**read_src**(*signature*, *cache_params*)
Lookup source code in disk cache and return file contents or None.

dijitso.cache.**report_cache_integrity**(*dmissing*, *out=<function warning>*)
Print cache integrity report.

dijitso.cache.**store_inc**(*signature*, *content*, *cache_params*)
Store header file within dijitso directories.

dijitso.cache.**store_log**(*signature*, *content*, *cache_params*)
Store log file within dijitso directories.

dijitso.cache.**store_src**(*signature*, *content*, *cache_params*)
Store source code in file within dijitso directories.

dijitso.cache.**write_library_binary**(*lib_data*, *signature*, *cache_params*)
Store compiled shared library from binary blob in numpy byte array to cache.

1.3.4 dijitso.cmdline module

This file contains the commands available through command-line dijitso-cache.

Each function cmd_<cmdname> becomes a subcommand invoked by:

```
dijitso-cache cmdname ...args
```

The docstrings in the cmd_<cmdname> are shown when running:

```
dijitso-cache cmdname --help
```

The 'args' argument to cmd_* is a Namespace object with the commandline arguments.

dijitso.cmdline.**args_checkout**(*parser*)

dijitso.cmdline.**args_clean**(*parser*)

dijitso.cmdline.**args_config**(*parser*)

dijitso.cmdline.**args_grep**(*parser*)

dijitso.cmdline.**args_grepfunction**(*parser*)

dijitso.cmdline.**args_show**(*parser*)

dijitso.cmdline.**args_version**(*parser*)

dijitso.cmdline.**cmd_checkout**(*args*, *params*)
copy files from cache to a directory

dijitso.cmdline.**cmd_clean**(*args*, *params*)
remove files from cache

dijitso.cmdline.**cmd_config**(*args*, *params*)
show configuration

dijitso.cmdline.**cmd_grep**(*args*, *params*)
grep content of header and source file(s) in cache

dijitso.cmdline.**cmd_grepfunction**(*args*, *params*)
search for function name in source files in cache

dijitso.cmdline.**cmd_show**(*args*, *params*)
show lists of files in cache

dijitso.cmdline.**cmd_version**(*args*, *params*)
print dijitso version

dijitso.cmdline.**parse_categories**(*categories*)

1.3.5 dijitso.jit module

This module contains the main jit() function and related utilities.

exception dijitso.jit.**DijitsoError**(*message*, *err_info*)
Bases: RuntimeError

dijitso.jit.**extract_factory_function**(*lib*, *name*)
Extract function from loaded library.

Assuming signature `(void *)()`, for anything else use look at ctypes documentation.

Returns the factory function or raises error.

dijitso.jit.**jit**(*jitable*, *name*, *params*, *generate=None*, *send=None*, *receive=None*, *wait=None*)
Just-in-time compile and import of a shared library with a cache mechanism.

A signature is computed from the name, params["generator"], and params["build"]. The name should be a unique identifier for the jitable, preferrably produced by a good hash function.

The signature is used to identity if the library has already been compiled and cached. A two-level memory and disk cache ensures good performance for repeated lookups within a single program as well as persistence across program runs.

If no library has been cached, the passed 'generate' function is called to generate the source code:

    header, source, dependencies = generate(jitable, name, signature, params["generator"])

It is expected to translate the 'jitable' object into C or C++ (default) source code which will subsequently be compiled as a shared library and stored in the disk cache. The returned 'dependencies' should be a tuple of signatures returned from other completed dijitso.jit calls, and are linked to when building.

The compiled shared library is then loaded with ctypes and returned.

For use in a parallel (MPI) context, three functions send, receive, and wait can be provided. Each process can take on a different role depending on whether generate, or receive, or neither is provided.

- Every process that gets a generate function is called a 'builder', and will generate and compile code as described above on a cache miss. If the function send is provided, it will then send the shared library binary file as a binary blob by calling send(numpy_array).
- Every process that gets a receive function is called a 'receiver', and will call 'numpy_array = receive()' expecting the binary blob with a compiled binary shared library which will subsequently be written to file in the local disk cache.
- The rest of the processes are called 'waiters' and will do nothing.
- If provided, all processes will call wait() before attempting to load the freshly compiled library from disk cache.

The intention of the above pattern is to be flexible, allowing several different strategies for sharing build results. The user of dijitso can determine groups of processes that share a disk cache, and assign one process per physical disk cache directory to write to that directory, avoiding multiple processes writing to the same files.

This forms the basis for three main strategies:

- Build on every process.
- Build on one process per physical cache directory.
- Build on a single global root node and send a copy of the binary to one process per physical cache directory.

It is highly recommended to avoid have multiple builder processes sharing a physical cache directory.

dijitso.jit.**jit_signature**(*name*, *params*)
Compute the signature that jit will use for given name and params.

1.3.6 dijitso.log module

dijitso.log.**set_log_level**(*level*)
Set verbosity of logging. Argument is int or one of "INFO", "WARNING", "ERROR", or "DEBUG".

dijitso.log.**get_logger**()

dijitso.log.**get_log_handler**()

dijitso.log.**set_log_handler**(*handler*)

1.3.7 dijitso.mpi module

Utilities for mpi features of dijitso.

dijitso.mpi.**bcast_uuid**(*comm*)
Create a unique id shared across all processes in comm.

dijitso.mpi.**create_comms_and_role**(*comm*, *comm_dir*, *buildon*)
Determine which role each process should take, and create the right copy_comm and wait_comm for the build strategy.

buildon must be one of "root", "node", or "process".

Returns (copy_comm, wait_comm, role).

dijitso.mpi.**create_comms_and_role_node**(*comm*, *node_comm*, *node_root*)
Approach: each node root builds, everyone waits on their node group.

dijitso.mpi.**create_comms_and_role_process**(*comm*, *node_comm*, *node_root*)
Approach: each process builds its own module, no communication.

To ensure no race conditions in this case independently of cache dir setup, we include an error check on the size of the autodetected node_comm. This should always be 1, or we provide the user with an informative message. TODO: Append program uid and process rank to basedir instead?

dijitso.mpi.**create_comms_and_role_root**(*comm*, *node_comm*, *node_root*)
Approach: global root builds and sends binary to node roots, everyone waits on their node group.

dijitso.mpi.**create_node_comm**(*comm*, *comm_dir*)
Create comms for communicating within a node.

dijitso.mpi.**create_node_roots_comm**(*comm*, *node_root*)
Build comm for communicating among the node roots.

dijitso.mpi.**create_subcomm**(*comm*, *ranks*)
Create a communicator for a set of ranks.

dijitso.mpi.**discover_path_access_ranks**(*comm*, *path*)
Discover which ranks share access to the same directory.

This cannot be done by comparing paths, because a path string can represent a local work directory or a network mapped directory, depending on cluster configuration.

Current approach is that each process touches a filename with its own rank in their given path. By reading in the filelist from the same path, we'll find which ranks have access to the same directory.

To avoid problems with leftover files from previous program crashes, or collisions between simultaneously running programs, we use a random uuid in the filenames written.

dijitso.mpi.**gather_global_partitions**(*comm*, *partition*)
Gather an ordered list of unique partition values within comm.

dijitso.mpi.**receive_binary**(*comm*)
Store shared library received as a binary blob to cache.

dijitso.mpi.**send_binary**(*comm*, *lib_data*)
Send compiled library as binary blob over MPI.

1.3.8 dijitso.params module

Utilities for dijitso parameters.

dijitso.params.**as_bool**(*value*)

dijitso.params.**as_str_tuple**(*p*)
Convert p to a tuple of strings, allowing a list or tuple of strings or a single string as input.

dijitso.params.**check_params_keys**(*default*, *params*)
Check that keys in params exist in defaults.

dijitso.params.**copy_params**(*params*)
Copy two-level dict of params.

dijitso.params.**default_build_params**()

dijitso.params.**default_cache_params**()

dijitso.params.**default_cxx_compiler**()
Default C++ compiler

dijitso.params.**default_cxx_debug_flags**()
Default C++ flags for debug=True. Note: FFC always overrides these.

dijitso.params.**default_cxx_flags**()
Default C++ flags for all build modes.

dijitso.params.**default_cxx_release_flags**()
Default C++ flags for debug=False. Note: FFC always overrides these.

dijitso.params.**default_generator_params**()

dijitso.params.**default_params**()

dijitso.params.**discover_config_filename**()

dijitso.params.**merge_params**(*default*, *params*)
Merge two-level param dicts.

dijitso.params.**read_config_file**()
Read config file and cache the contents for the duration of the process.

dijitso.params.**session_default_params**()

dijitso.params.**validate_params**(*params*)
Validate parameters to dijitso and fill in with defaults where missing.

1.3.9 dijitso.signatures module

dijitso.signatures.**canonicalize_params_for_hashing**(*params*)

dijitso.signatures.**hash_params**(*params*)

dijitso.signatures.**hashit**(*data*)
Return hash of anything with a repr implementation.

1.3.10 dijitso.str module

String compatibility utilities.

dijitso.str.**as_unicode**(*s*)
Return s if unicode string, or decode bytes to unicode string using utf-8.

1.3.11 dijitso.system module

Utilities for interfacing with the system.

dijitso.system.**get_status_output**(*cmd*, *input=None*, *cwd=None*, *env=None*)
Replacement for commands.getstatusoutput which does not work on Windows (or Python 3).

dijitso.system.**gunzip_file**(*gz_filename*)
Gunzip a file.

dijitso.system.**gzip_file**(*filename*)
Gzip a file.

New file gets .gz extension added.

Does nothing if the .gz file already exists.

Original file is never touched.

dijitso.system.**ldd**(*libname*)
Run the ldd system tool on libname.

Returns output as a dict {basename: fullpath} with all dynamic library dependencies and their resolution path.

This is a debugging tool and may fail if ldd is not available or behaves differently on this system.

dijitso.system.**lockfree_move_file**(*src*, *dst*)
Lockfree and portable nfs safe file move operation.

If target filename exists with different content, will move it to filename.old and emit a warning.

Taken from textual description at http://stackoverflow.com/questions/11614815/a-safe-atomic-file-copy-operation

dijitso.system.**make_dirs**(*path*)
Creates a directory (tree).

Ignores error if the directory already exists.

dijitso.system.**make_executable**(*filename*)
Make script executable by setting user eXecutable bit.

dijitso.system.**move_file**(*srcfilename*, *dstfilename*)
Move or copy a file.

dijitso.system.**read_textfile**(*filename*)
Try to read file content, if necessary unzipped from filename.gz, return None if not found.

dijitso.system.**rename_file**(*src*, *dst*)
Rename a file.

Ignores error if the destination file exists.

dijitso.system.**store_textfile**(*filename*, *content*)
Store content to filename without race conditions.

Works by first writing to a unique temp file and then moving to final destination.

Handles both bytes and unicode.

dijitso.system.**try_copy_file**(*src*, *dst*)
Try to copy a file.

NB! Ignores any error.

dijitso.system.**try_delete_file**(*filename*)
Try to remove a file.

Ignores error if filename doesn't exist.

dijitso.system.**try_rename_file**(*src*, *dst*)
Try to rename a file.

NB! Ignores error if the SOURCE doesn't exist or the destination already exists.

1.3.12 Module contents

dijitso.**validate_params**(*params*)
Validate parameters to dijitso and fill in with defaults where missing.

dijitso.**jit**(*jitable*, *name*, *params*, *generate=None*, *send=None*, *receive=None*, *wait=None*)
Just-in-time compile and import of a shared library with a cache mechanism.

A signature is computed from the name, params["generator"], and params["build"]. The name should be a unique identifier for the jitable, preferrably produced by a good hash function.

The signature is used to identity if the library has already been compiled and cached. A two-level memory and disk cache ensures good performance for repeated lookups within a single program as well as persistence across program runs.

If no library has been cached, the passed 'generate' function is called to generate the source code:

header, source, dependencies = generate(jitable, name, signature, params["generator"])

It is expected to translate the 'jitable' object into C or C++ (default) source code which will subsequently be compiled as a shared library and stored in the disk cache. The returned 'dependencies' should be a tuple of signatures returned from other completed dijitso.jit calls, and are linked to when building.

The compiled shared library is then loaded with ctypes and returned.

For use in a parallel (MPI) context, three functions send, receive, and wait can be provided. Each process can take on a different role depending on whether generate, or receive, or neither is provided.

- Every process that gets a generate function is called a 'builder', and will generate and compile code as described above on a cache miss. If the function send is provided, it will then send the shared library binary file as a binary blob by calling send(numpy_array).
- Every process that gets a receive function is called a 'receiver', and will call 'numpy_array = receive()' expecting the binary blob with a compiled binary shared library which will subsequently be written to file in the local disk cache.
- The rest of the processes are called 'waiters' and will do nothing.
- If provided, all processes will call wait() before attempting to load the freshly compiled library from disk cache.

The intention of the above pattern is to be flexible, allowing several different strategies for sharing build results. The user of dijitso can determine groups of processes that share a disk cache, and assign one process per physical disk cache directory to write to that directory, avoiding multiple processes writing to the same files.

This forms the basis for three main strategies:

- Build on every process.
- Build on one process per physical cache directory.
- Build on a single global root node and send a copy of the binary to one process per physical cache directory.

It is highly recommended to avoid have multiple builder processes sharing a physical cache directory.

dijitso.**extract_factory_function**(*lib*, *name*)
Extract function from loaded library.

Assuming signature `(void *)()`, for anything else use look at ctypes documentation.

Returns the factory function or raises error.

dijitso.**set_log_level**(*level*)
Set verbosity of logging. Argument is int or one of "INFO", "WARNING", "ERROR", or "DEBUG".

1.4 Release notes

1.4.1 Changes in the next release

Summary of changes

Note: Developers should use this page to track and list changes during development. At the time of release, this page should be published (and renamed) to list the most important changes in the new release.

- Remove Python 2 support. Now Python 3.5 or later is required.

Detailed changes

Note: At the time of release, make a verbatim copy of the ChangeLog here (and remove this note).

1.4.2 Changes in version 2018.1.0

dijitso 2018.1.0 was released on 2018-06-14.

- Very minor cleanups.

1.4.3 Changes in version 2017.1.0.post1

dijitso 2017.1.0.post1 was released on 2017-09-12.

- Change PyPI package name to fenics-dijitso.

1.4.4 Changes in version 2017.1.0

dijitso 2017.1.0 was released on 2017-05-09.

- Minor fixes

1.4.5 Changes in version 2016.2.0

dijitso 2016.2.0 was released on 2016-11-30.

- Introduce commandline script 'dijitso' with various subcommands to interact with the cache
- Improve extraction of source files to reproduce compilation failure during jit
- Implement support for linking between jit modules
- Add optional dependency on subprocess32 to handle fork safety on infiniband clusters
- Remove instant dependency

1.4.6 Changes in version 2016.1.0

dijitso 2016.1.0 was released on 2016-06-23.

- This is the first version that covers all the immediate needs of the FEniCS Form Compiler (FFC) for just in time compilation of generated code using ctypes to import a simple generated factory function.

[FIXME: These links don't belong here, should go under API reference somehow.]

- genindex
- modindex

Python Module Index

d

Index